Solving the Selective Multi-Category Parallel-Servicing Problem

Range, Troels Martin; Lusby, Richard Martin ; Larsen, Jesper

Publication date:
2013

Document Version
Publisher's PDF, also known as Version of record

Solving the Selective Multi-Category Parallel-Servicing Problem

by

Troels Martin Range,
Richard Martin Lusby
and
Jesper Larsen

Discussion Papers on Business and Economics
No. 5/2013

FURTHER INFORMATION
Department of Business and Economics
Faculty of Social Sciences
University of Southern Denmark
Campusvej 55
DK-5230 Odense M
Denmark

Tel.: +45 6550 3271
Fax: +45 6550 3237
E-mail: lho@sam.sdu.dk
http://www.sdu.dk/ivoe

ISBN 978-87-91657-83-2

Solving the Selective Multi-Category Parallel-Servicing Problem

Troels Martin Range

Department of Business and Economics, COHERE, University of Southern Denmark,
Campusvej 55, 5230 Odense M, Denmark,
tra@sam.sdu.dk

Richard Martin Lusby

Department of Engineering Management, Technical University of Denmark
Produktionstorvet, building 426, 2800 Kgs. Lyngby, Denmark,
rmlu@dtu.dk

Jesper Larsen

Department of Engineering Management, Technical University of Denmark
Produktionstorvet, building 426, 2800 Kgs. Lyngby, Denmark,
jesla@dtu.dk

March 1, 2013

Abstract

In this paper we present a new scheduling problem and describe a shortest path based heuristic as well as a dynamic programming based exact optimization algorithm to solve it. The Selective Multi-Category Parallel-Servicing Problem (SMCPSP) arises when a set of jobs has to be scheduled on a server (machine) with limited capacity. Each job requests service in a prespecified time window and belongs to a certain category. Jobs may be serviced partially, incurring a penalty; however, only jobs of the same category can be processed simultaneously. One must identify the best subset of jobs to process in each time interval of a given planning horizon while respecting the server capacity and scheduling requirements. We compare the proposed solution methods with a MILP formulation and show that the dynamic programming approach is faster when the number of categories is large, whereas the MILP can be solved faster when the number of categories is small.

Keywords: Machine scheduling, dynamic programming, node-disjoint shortest-path problem, preprocessing.

JEL Code: C61, **MSC Code:** 90B35, 90C35

1 Introduction

Given a set of possible jobs which can be undertaken in the future, one would like to *select* a subset of these jobs which maximizes the total profit, and this must be done within the bounds of the available resources. Such a problem is referred to as a selection problem. Selection problems often arise as pricing problems when using column generation. Examples are Gilmore and Gomory [1961], who use a knapsack problem as the pricing problem in column generation for solving the cutting stock problem, and Desrochers et al. [1992], who solve the vehicle routing problem with time windows by decomposing the problem into a set partitioning problem as the master problem and a shortest path problem with time windows. Chen and Powell [1999] apply column generation to the parallel machine scheduling problem where the pricing problem selects a sequence of jobs to schedule on a single machine.

In this paper we describe and develop methods for a specific selection problem, which we have termed the Selective Multi-Category Parallel-Servicing Problem (SMCPSP). This problem has a set of jobs, where each job belongs to a category and has a prespecified service time interval during which it requests service. It is only possible to service jobs from the same category simultaneously, and it is only possible to service a limited number of jobs at the same time. The planning period is split into atomic time intervals, and servicing each job in an atomic period yields a direct profit. However, it is possible to service only a fraction of a job, but doing this will add a penalty cost. The problem is then to identify the most profitable subset of jobs to service in each atomic period for the full planning period such that only jobs from the same category are serviced in the same period and such that the limited capacity of the server is not violated. Note that in this paper the term server is synonymous with *machine*. Instead of maximizing profit, we prefer to minimize the penalty cost, where a positive profit corresponds to a negative penalty cost.

To illustrate this, we provide a solution to an instance of the problem in Figure 1. The instance has 500 potential jobs divided into 16 categories, 25 periods, and a server capacity of four. The periods are given horizontally, whereas the positions within the server are illustrated vertically. The jobs undertaken are shown as blocks. If a block has either a gray start or end, then it is partially serviced.

One particular application of the SMCPSP is the pricing problem for a room based decomposition of the Patient Admission Scheduling problem (PAS) introduced by Demeester et al. [2010]. This decomposition was proposed by Range et al. [2013]. The PAS is the problem of assigning patients to rooms during a planning period such that patients are assigned to hospital rooms in the best possible manner, while being transferred in and out of rooms as little as possible. In this problem patients correspond to jobs, while the categories correspond to the genders of the patients (which cannot be mixed). One can identify rooms as servers, where each room has a certain capacity (i.e. the number of beds). In the PAS problem, since a patient may be transferred in and out of rooms during their admission, he/she can be viewed as a job which can be partially serviced by any of his/her compatible rooms. While it is possible to admit a patient to a room for a fraction of their stay, it is not possible to admit a patient to the hospital for fewer days than the patient requests. Thus, if a patient is assigned to a room for part of their stay, they must be admitted to at least one other room for the remainder of their stay to *complete* the service.

This paper is organized as follows. Initially, Section 2 provides a more formal definition of the problem and presents a mathematical model thereof. Given that this is a new problem, we review related literature in Section 3 to help place this new scheduling problem. Section 4 introduces two underlying networks for the problem and discusses several shortest path based

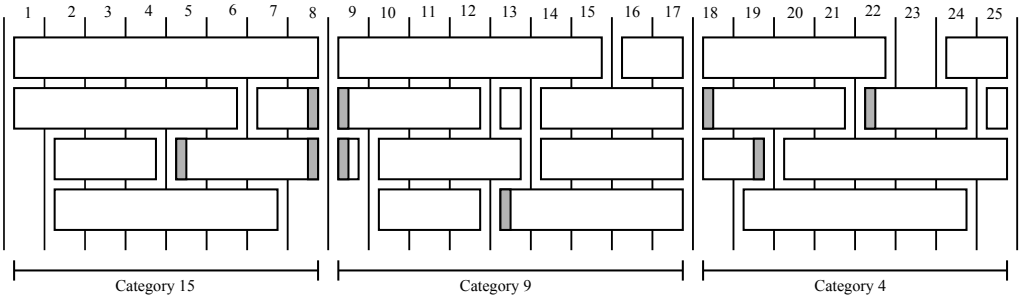

Figure 1: Solution to instance 14 with 16 categories and a server capacity of four.

methods for generating lower and upper bounds on the SMCPSP. To reduce the problem size, we describe efficient preprocessing strategies in Section 5. We exploit the preprocessed problem to construct an exact dynamic programming algorithm in Section 6. Computational results are presented in Section 7, while concluding remarks are given in Section 8.

2 Problem Description

A server can process Q jobs of the same category simultaneously. A set of possible jobs, $\mathcal{J} = \{1, \ldots, J\}$, is given where each job requests service in a predefined period. Time is discretized into atomic periods, $\mathcal{T} = \{1, \ldots, T\}$. Hence, each job, $j \in \mathcal{J}$, has a starting period, $a_j \geq 1$, and an ending period, $b_j \leq T$, and we define the set of service periods for job j as $\mathcal{T}_j = \{a_j, \ldots, b_j\}$. We denote the set of all jobs which may be serviced in period t as $\mathcal{J}_t^{per} = \{j \in \mathcal{J} | t \in \mathcal{T}_j\}$. Processing job j in period t has a penalty (or cost) of $p_{jt} \in \mathbb{R} \cup \{\infty\}$, where $p_{jt} = \infty$ for $t \notin \mathcal{T}_j$. If the penalty is negative, then we interpret the penalty as a profit. A set of categories, $\mathcal{C} = \{1, \ldots, C\}$, is given, and $c_j \in \mathcal{C}$ is the category of job $j \in \mathcal{J}$. Furthermore, we let $\mathcal{J}_c = \{j \in \mathcal{J} | c_j = c\}$ be the subset of jobs belonging to category $c \in \mathcal{C}$. As it is allowed to partially service a full job, we define two penalties. The first penalty, π^s, is a penalty for starting (or restarting) a job later than a_j. The second penalty, π^e, is for prematurely ending the service of a job, i.e. stop processing an already started job before period b_j. These penalties are equal for all jobs and all periods. To ease the notation, we define the set $\mathcal{W} = \{(j,t) \in \mathcal{J} \times \mathcal{T} | t \in \mathcal{T}_j\}$ as the set of feasible job-period combinations. Furthermore, we let the set $\mathcal{J}_{ct} = \mathcal{J}_c^{cat} \cap \mathcal{J}_t^{per}$ be the jobs of category c which can be undertaken in period t. The server can only process jobs from the same category in a single period, i.e. two jobs, $i, j \in \mathcal{J}$ with $c_j \neq c_i$, cannot be processed in the same period. We let the set $\mathcal{K} = \{(j,i) \in \mathcal{J} \times \mathcal{J} | c_j \neq c_i\}$ be the pairs of jobs which are incompatible with each other.

A solution to the SMCPSP is a sequence of sets of jobs $\mathcal{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_T)$, where $\mathcal{S}_t \subseteq \mathcal{J}$. We say that a solution is feasible for the SMCPSP if the three following conditions are all satisfied:

1. For all $t \in \mathcal{T}$ the capacity constraint is satisfied, i.e. $|\mathcal{S}_t| \leq Q$.
2. For all $t \in \mathcal{T}$ the set $\mathcal{S}_t \subseteq \mathcal{J}_t^{per}$.
3. For all $t \in \mathcal{T}$ either the set $\mathcal{S}_t = \emptyset$ or the set $\mathcal{S}_t \subseteq \mathcal{J}_c^{cat}$ for at most one category $c \in \mathcal{C}$.

The set of all feasible sets in period $t \in \mathcal{T}$ is denoted $\mathcal{F}_t$. The size of $\mathcal{F}_t$ is the set of possible combinations of jobs in period t, which is given by $\sum_{c\in\mathcal{C}} \sum_{q=0}^{Q} \binom{|\mathcal{J}_{ct}|}{q}$. This is exponential in the capacity, Q, as well as in the size, $|\mathcal{J}_{ct}|$, of the possible jobs within the same categories.

We let $f_{t-1,t}(\mathcal{S}_1, \mathcal{S}_2)$ be a function counting the number of late starts for the set of jobs $\mathcal{S}_2$ in period t if we have processed the jobs in $\mathcal{S}_1$ in period $t-1$. We can express the function explicitly as

$$f_{t-1,t}(\mathcal{S}_1, \mathcal{S}_2) = \left|(\mathcal{S}_2 \cap \mathcal{J}_{t-1}^{per}) \setminus \mathcal{S}_1\right| \tag{1}$$

Similarly, we let $g_{t-1,t}(\mathcal{S}_1, \mathcal{S}_2)$ be a function counting the number of prematurely ended jobs for the set of jobs $\mathcal{S}_2$ when we have processed the jobs in $\mathcal{S}_1$ in period $t-1$. The number of jobs prematurely ended can be expressed as

$$g_{t-1,t}(\mathcal{S}_1, \mathcal{S}_2) = |(\mathcal{S}_1 \cap \mathcal{J}_t^{per}) \setminus \mathcal{S}_2| \tag{2}$$

Then we can express the objective function as

$$P(\mathcal{S}) = \sum_{t\in\mathcal{T}} \sum_{j\in\mathcal{S}_t} p_{jt} + \pi^s \sum_{t\in\mathcal{T}\setminus\{1\}} f_{t-1,t}\left(\mathcal{S}_{t-1}, \mathcal{S}_t\right) + \pi^e \sum_{t\in\mathcal{T}\setminus\{1\}} g_{t-1,t}\left(\mathcal{S}_{t-1}, \mathcal{S}_t\right) \tag{3}$$

We say that a solution, $\mathcal{S}$, with cost $P(\mathcal{S})$ is minimal if we cannot remove a full job or parts of a job without changing the cost and if we cannot exchange parts of a job with parts of another job yielding the same cost but requiring less processing time. The first condition avoids having jobs in the solution with zero penalty, and the second ensures as many free periods as possible. In the case where the solution is not minimal, we have alternative solutions to the problem. The selective multi-category parallel-servicing problem is then the problem of identifying a feasible solution, $\mathcal{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_T)$, which minimizes $P(\mathcal{S})$. The optimal solution value is denoted P^*.

It should be noted that if $\pi^s = \pi^e = 0$, then the problem reduces to the problem of identifying the most profitable job combination in each period and each category as there would be no cost component connecting the periods. Hence, for each period we could just select jobs with positive profit in decreasing order in each category and then select the category which has the highest profit. It is easy to see that this solution would be optimal.

On the other hand if, $\pi^s = \pi^e = \infty$, then we disallow partially servicing a job and essentially turn this problem into a unit-weight multiple-choice multiple-knapsack problem with precedence constraints. The unit weight is because the weight of a job in each of the capacity constraints is one unit. It is a multiple-knapsack problem because the problem has a multiple number of knapsack constraints corresponding to the capacity constraints. For each knapsack constraint we can only choose jobs of the same category, which makes the problem a multiple choice problem. Finally, if it is not allowed to partially service a job, once servicing the job has begun, it will continue to be serviced in subsequent periods (assuming the job has a service time of more than one time period). This gives (possibly cyclic) precedence constraints.

2.1 A Linear Integer Programming Model

A mathematical model can be constructed for the SMCPSP. We use the following variables. Let $x_{jt} \in \{0, 1\}$ indicate whether or not job j is undertaken in period t. Furthermore, let γ_{jt}^s indicate that job j is started or restarted in period t, and γ_{jt}^e indicate that job j is either ended or preempted from period $t-1$ to period t. To ensure that only a single category is

used in each period t, we let y_{ct} indicate whether or not a job in category c is undertaken in period t. A mathematical formulation of the SMCPSP is then

$$\min \sum_{(j,t)\in\mathcal{W}} \left(p_{jt}x_{jt} + \pi^s\gamma^i_{jt} + \pi^e\gamma^e_{jt}\right) \tag{4}$$

$$\text{s.t.} \sum_{j\in\mathcal{J}^{per}_t} x_{jt} \leq Q, \qquad \forall\, t \in \mathcal{T} \tag{5}$$

$$x_{jt} - x_{j,t-1} - \gamma^i_{jt} \leq 0, \qquad \forall j \in \mathcal{J}, \forall t \in \mathcal{T}_j \setminus \{a_j\} \tag{6}$$

$$x_{jt} - x_{j,t+1} - \gamma^e_{jt} \leq 0, \qquad \forall j \in \mathcal{J}, \forall t \in \mathcal{T}_j \setminus \{b_j\} \tag{7}$$

$$x_{jt} - y_{ct} \leq 0, \qquad \forall c \in \mathcal{C}, \forall j \in \mathcal{J}^{cat}_c, \forall t \in \mathcal{T} \tag{8}$$

$$\sum_{c\in\mathcal{C}} y_{ct} \leq 1, \qquad \forall t \in \mathcal{T} \tag{9}$$

$$x_{jt} \in \{0,1\}, \qquad \forall (j,t) \in \mathcal{W} \tag{10}$$

$$\gamma^i_{jt}, \gamma^e_{jt} \geq 0, \qquad \forall (j,t) \in \mathcal{W} \tag{11}$$

$$y_{ct} \geq 0, \qquad \forall c \in \mathcal{C}, \forall t \in \mathcal{T} \tag{12}$$

The objective (4) minimizes the total penalty of the solution. Constraints (5) state that the capacity in each period cannot be exceeded. Constraints (6) and (7) ensure that the start and end are measured respectively, while constraints (8) state that category c is used in period t if job j from category c is undertaken in period t. Constraints (9) make the categories mutually exclusive by only allowing a single category to be active in each period. Finally, constraints (10)-(12) set the domains of the variables. Note that if the x_{jt}-variables are all binary, then the remaining variables will also have binary values in an optimal solution. Hence, these are just standard non-negative continuous variables.

3 Related Literature

The problem considered in this paper requires that a subset of jobs is *selected* (all of which belong to the same category) to be serviced on each day of a specified planning horizon without exceeding the capacity of the server. In this section we briefly review other similar selection problems and present a brief discussion on job shop scheduling.

Selection problems typically entail finding an optimal subset of items from an extremely large number of possibilities and are, not surprisingly, widely studied in the operations research literature. In the vehicle routing problem (VRP), for instance, one has a fleet of vehicles and must determine which subsets of customers should be visited by which vehicles to minimize the total driving distance while respecting a number of different constraints (see e.g. Drexl [2012]). Similarly, in the crew pairing problem faced by airlines, one must determine an optimal subset of flights to be flown by each crew member (see e.g. Anbil et al. [1993], Andersson et al. [1998]). Typically, the selection problem for each vehicle/crew member appears as the subproblem in a column generation framework and is solved as a resource-constrained shortest-path problem. For a general introduction to this, we refer the reader to Irnich and Desaulniers [2005]. This problem involves computing a least cost shortest path that respects a set of resource constraints. It is typically solved using a label setting algorithm, which implicitly considers all paths in the network and relies on strong

dominance criteria to remove unpromising labels and reduce the computation time (see e.g. Irnich [2008]).

Other similar selection problems include the traveling salesman problem with profits (also termed the Selective traveling salesman problem), the prize collecting steiner tree problem, and the prize collecting arc routing problem. The former is a variant of the well-known traveling Salesman problem in which it is no longer necessary to visit all customers. Instead, a profit is assigned to each vertex, and one must maximize the collected profit, balanced with travel cost. A survey on variants of this selection problem as well as techniques for solving them is given by Feillet et al. [2005]. Similarly, in the prize collecting steiner tree problem each vertex of a weighted graph is associated with a prize and a cost, and one must find a spanning tree by using a subset of the nodes that collects a total prize not less than a certain quota. Haouri et al. [2013] present several compact mixed integer programming formulations and describe the theoretical properties of each. Near-optimal solutions for instances containing up to 2500 nodes and 3125 edges are given. In the prize collecting arc routing problem, a prize is assigned to each edge (in addition to cost) and is collected the first time the edge is traversed. One must typically construct a tour in the given network that maximizes the sum of the collected prizes minus the travel cost. The polyhedral structure of this problem is described in Aráoz et al. [2006], while Black et al. [2013] consider an extension of the problem in which the arc prizes are time dependent.

The SMCPSP with $C = 2$ arises as the pricing problem of the decomposition of the Patient Admission Scheduling (PAS) problem. In PAS the server corresponds to a room having Q identical beds, and the categories are the genders which in some cases cannot be mixed. The jobs are given by patients having to occupy a bed during their stay at the hospital, and the penalties correspond to the contribution to the reduced cost coefficient for a room schedule. Finally, penalties π^s and π^e are incurred whenever we transfer a patient into the room after their admission and transfer a patient out of the room before his/her discharge. PAS was introduced by Demeester et al. [2010], whereas the decomposition is described by Range et al. [2013].

Another well-known selection problem is the knapsack problem (see e.g. Pisinger [1995]). In this problem one is given a set of items, each with an associated weight and profit, and one must select an optimal subset of items that maximizes the profit while respecting a certain weight capacity. One can view the SMCPSP as a series of interconnected knapsack problems where the weight of each item (job) is identical and the capacity of each knapsack is simply the capacity of the server. In addition to the capacity requirement, one must also ensure that all items (jobs) placed in the knapsack belong to the same category. Such restrictions require additional constraints and hence give rise to a *multidimensional knapsack*. A recent review of knapsack problem variants and their relative difficulty is provided by Smith-Miles and Lopes [2012].

Since this problem entails assigning jobs to a machine, this section would not be complete without a short discussion of the SMCPSP's relation to job-shop scheduling. In job-shop scheduling problems one is given a set of jobs where each job requires a certain processing time on a set of machines that can host one job at a time. Typically, one must schedule the jobs in a way that minimizes the total time required to complete the jobs, the tardiness of the jobs, or the idle time of the machines. Often no preemption is permitted. That is, one cannot end a job prematurely once it has begun. There are several differences between the job-shop scheduling problem and the SMCPSP. In the latter, all jobs require servicing by one machine, jobs cannot be processed outside their service window, the time to complete all jobs is known a priori, the server can host more than one job at a time, and preemption

is possible. An introduction to job-shop scheduling and its complexity is given in Garey [1976].

In the case where the problem only has a single category, the problem can be solved as a Q node-disjoint shortest-path problem, i.e. identifying Q paths having no nodes in common such that the total sum of the penalties obtained is minimized. The network required for this is described in Section 4.2. Bang-Jensen and Gutin [2001] include a chapter on identification of disjoint paths in digraphs along with a description of the relevant literature. In our case, however, Q node-disjoint paths always exist, but we are interested in the least total penalty instead. Tholey [2005] develops a polynomial algorithm for finding two node disjoint paths in a directed acyclic graph. Suurballe and Tarjan [1984] construct a dynamic programming algorithm for identifying minimum total cost pairs of paths which are edge disjoint in a graph with non-negative costs. The emphasis in our case is somewhat different as we are trying to identify node-disjoint paths in a graph having possibly negative costs.

4 Underlying networks

Before solving the problem, we discuss two networks which are closely related to the problem. The first network is a small network representing whether or not a single job is processed in each of the time periods. We will refer to this network as the *single job network* and it is described in Section 4.1. The single job network is primarily used for preprocessing (see Section 5.1) and as reference notation. The second network is a full representation of a single stream of jobs, i.e. given that we have a server with a capacity of one job in each period, the network gives the possible sequences of (partial) jobs which can be undertaken in each period. This network can be used to identify upper and lower bounds of an optimal solution (see Section 4.3). We refer to this network as the *full period-job network* and it is described in Section 4.2. We finish this section with a few comments on solving shortest path problems in acyclic graphs (see Section 4.4) since our networks are acyclic due to the temporal nature of the scheduling environment.

4.1 Single job network

Given a job, $j \in \mathcal{J}$, we will only include the full job or part of the job in an optimal solution if it reduces the objective. This is because it is allowed to exclude part of or the full job in the solution without incurring a penalty. To exploit this, we set up a directed graph which can be used to calculate the smallest possible penalty for a job.

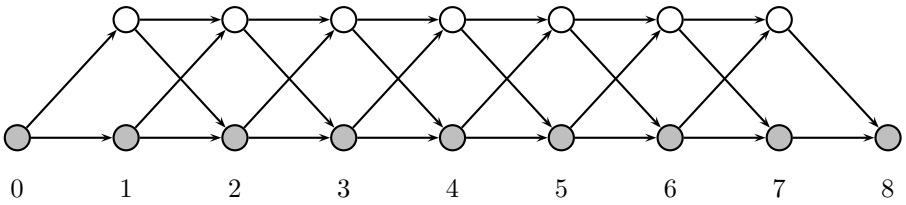

Figure 2: Single job graph for $\mathcal{T} = \{1, \ldots, 7\}$

The graph is illustrated in Figure 2. It has two nodes for each $t \in \mathcal{T}$, where the first node corresponds to the job being processed in period t, while the second node corresponds

to the job not being processed in period t. We let $V = \{v_t | t \in \mathcal{T}\}$ be the set of nodes for jobs being processed (the white nodes) and $U = \{u_t | t \in \mathcal{T} \cup \{0, T+1\}\}$ be the set of nodes for the job not being processed (the gray nodes). Additionally, we have a source node, u_0, and a sink node, u_{T+1}, as a part of U since these correspond to the case where the job is not being processed prior to the planning horizon or after the planning horizon. In all, we have the node set $N = V \cup U$. The graph has an arc from each node corresponding to period t to each node corresponding to period $t+1$ for $t = 0, \ldots, T$. An arc (u_t, v_{t+1}) between a node from $u_t \in U$ to a node from $v_{t+1} \in V$ indicates that the job is started in period $t+1$. Likewise, an arc (v_t, u_{t+1}) indicates that the job is discontinued after period t. We denote the set of arcs A. This is the underlying graph for any job in the problem. The only difference between the jobs is how the penalties are set on the arcs in the graph. Thus, we will set up an individual cost matrix for each of the jobs. We will denote this cost matrix as $\mathbf{c}^j$ for job j, and it can be specified as follows:

$$c^j_{hk} = \begin{cases} p_{j,t+1}, & (h,k) = (v_t, v_{t+1}) \\ 0, & (h,k) = (u_t, u_{t+1}) \\ p_{j,t+1} + \pi^s, & (h,k) = (u_t, v_{t+1}), t+1 > a_j \\ p_{j,t+1}, & (h,k) = (u_t, v_{t+1}), t+1 \leq a_j \\ \pi^e, & (h,k) = (v_t, u_{t+1}), t < b_j \\ 0, & (h,k) = (v_t, u_{t+1}), t \geq b_j \\ \infty, & (h,k) \notin A \end{cases} \tag{13}$$

The single job graph for job j is then denoted $G_j(N, A, \mathbf{c}^j)$.

A path through this graph corresponds to a schedule for processing the job. Such a schedule can be divided into subsequences of periods where each subsequence either processes the job for the entire subsequence or does not process the job for the entire subsequence. In the following discussion we denote a subpath $(u_t, v_{t+1}, \ldots, v_{s-1}, u_s)$ for a *processing sequence* for $t < s-1$. Similarly, we denote the subpath $(u_t, u_{t+1}, \ldots, u_{s-1}, u_s)$ as a *non-processing sequence* for $t < s$. The cost of using a non-processing sequence is always zero. Any path can be composed of a sequence of processing sequences and non-processing sequences having identical start and end nodes. Let α_{jts} be the cost of traversing this processing sequence in the graph for job j. If $\alpha_{jts} \geq 0$, then the processing sequence will yield a non-negative penalty, and it is therefore better to leave the job unprocessed. In other words, if $\alpha_{jts} \geq 0$, then it is better to use the non-processing sequence $(u_t, u_{t+1}, \ldots, u_{s-1}, u_s)$ instead of the path $(u_t, v_{t+1}, \ldots, v_{s-1}, u_s)$ as it makes it possible to process another job, i, for the processing sequence which may have negative α_{its}. Likewise, a *partial processing sequence* is a subpath $(v_t, v_{t+1}, \ldots, v_{s-1}, u_s)$ starting with a processing node and having only a non-processing node as the last node. For a given job $j \in \mathcal{J}$, we let β_{jts} be the penalty for the partial processing sequence starting in node v_t and ending in node u_s. Note that $\beta_{jts} + c^j_{u_{t-1}, v_t} = \alpha_{j,t-1,s}$. Figure 3 schematically shows this. The full black path represents the processing sequence $(u_2, \ldots, u_7)$ with length $\alpha_{j,2,7}$, while the dashed path indicates the partial processing sequence $(v_2, \ldots, u_7)$ with length $\beta_{j,2,7}$. Obviously, if we extend the dashed path by the dotted arc (u_1, v_2) with cost $c^j_{u_1,v_2}$, we would obtain the processing sequence $(u_1, \ldots, u_7)$ with length $\alpha_{j,1,7}$.

For two nodes, $n_1, n_2 \in N$, corresponding to job j for the graph $G_j(N, A, \mathbf{c}^j)$, we let $\delta_j(n_1, n_2)$ be the value of the shortest path from n_1 to n_2. If no path exists from n_1 to n_2, then we define $\delta_j(n_1, n_2) = \infty$. In the case where a path exists between two non-processing nodes, $u_t, u_s \in U$ with $t < s$, a trivial upper bound on the shortest path is obtained by using the path $(u_t, \ldots, u_s)$ which has value 0, i.e. for each pair u_t, u_s with $t < s$ we have

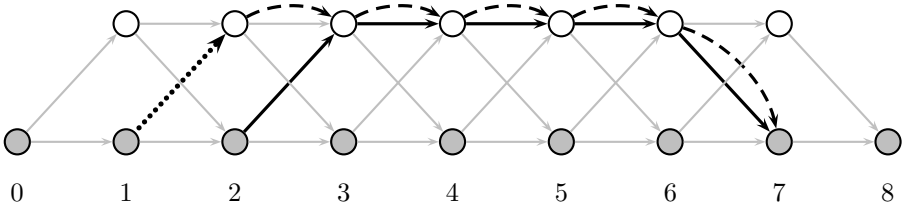

Figure 3: Relationship between α_{jts} and β_{jts}

that $\delta_j(u_t, u_s) \leq 0$ for all $j \in \mathcal{J}$. Next, for a processing node v_t, the value $\delta_j(v_t, u_s)$ is the best possible value for finishing job j before period s if it is being processed in period t.

Given that there are two (sub-)paths in the single job network, we can compare these in at least two ways. One way is to compare the cost of the two paths – the one with the lowest cost is deemed better than the other path. The second way of comparing the paths is on the utilization of the server. Intuitively, if one path utilizes the server less than the other path, then the first path is considered to be better, as it allows for other paths to utilize the server more. To be more precise, let $\mathcal{P} = (w_t, \ldots, w_s)$ and $\mathcal{P}' = (w'_t, \ldots, w'_s)$ be two paths, with $t < s$, and where $w_t = w'_t$ and $w_s = w'_s$. We say that $\mathcal{P}$ is more flexible than $\mathcal{P}'$ if $\{w \in \mathcal{P}' | w \in U\} \subset \{w \in \mathcal{P} | w \in U\}$. A more flexible path does not utilize the server in periods where the less flexible path does not utilize the server. Furthermore, the more flexible path has some periods where it does not utilize the server but where the less flexible path does. Hence, the more flexible path allows other paths to utilize the server in these periods. Note that the processing sequence $(u_t, v_{t+1}, \ldots, v_{s-1}, u_s)$ is the least flexible (u_t, u_s)-path, i.e. any other (u_t, u_s)-path in the graph will be more flexible than the processing sequence. On the other hand, the non-processing sequence $(u_t, u_{t+1}, \ldots, u_{s-1}, u_s)$ is the most flexible (u_t, u_s)-path. This is an important feature of the processing sequences and non-processing sequences, which we will exploit in Propositions 3-5.

4.2 Full Period-Job Network

In each period we can either undertake jobs or have no jobs being serviced. Let $E = \{e_t | t \in \mathcal{T}\}$ be the set of nodes indicating that no job is undertaken in the period $t \in \mathcal{T}$. For each job-time combination in $\mathcal{W}$, a node is added and we refer to an element of this set as $w_{jt} \in \mathcal{W}$. Finally, a source node, o, and a sink node, d, are used. This gives the node set $N = \mathcal{W} \cup E \cup \{o, d\}$.

The graph is layered and has a layer for each period $t \in \mathcal{T}$ as well as a layer for the source node and a layer for the sink node. We label the layer for the source node as 0 and label the layer for the sink node as $T + 1$. An arc exists for each pair of nodes in adjacent layers in the direction of increasing layers. The cost of an arc (h, k) depends on three factors: (1) if $k = w_{j,t+1} \in \mathcal{W}$, then we put $c^{pen}_{hk} = p_{j,t+1}$, otherwise we put $c^{pen}_{hk} = 0$; (2) if $h = w_{jt} \in \mathcal{W}$ with $t < b_j$ and $k \neq w_{j,t+1} \in \mathcal{W}$, then job j is finished prematurely and we put $c^{out}_{hk} = \pi^e$, otherwise we put $c^{out}_{hk} = 0$; and (3) if $k = w_{j,t+1} \in \mathcal{W}$ with $t + 1 > a_j$ and $h \neq w_{jt} \in \mathcal{W}$, then job j is started late and we put $c^{in}_{hk} = \pi^s$, otherwise we put $c^{in}_{hk} = 0$. The cost of arc (h, k) is then $c_{hk} = c^{pen}_{hk} + c^{out}_{hk} + c^{in}_{hk}$.

4.3 Shortest Path Based Bounds

It is possible to identify simple bounds on the problem by observing that a path through the full period-job network described in Section 4.2 yields a single stream of jobs to be executed. Hence, solving a shortest (o, d)-path problem in the full period-job network yields a least cost sequence of nodes having one node for each layer and where the nodes either correspond to a job being processed in the period or the server being idle for a single stream in the period. If $Q = 1$, then this is clearly an optimal solution for the SMCPSP. However, this is not the case for $Q \geq 2$, but we can use the graph to obtain upper and lower bounds on the optimal solution value for SMCPSP.

Let $z(M)$ be the value of the shortest (o, d)-path in the subgraph induced by $M \subseteq N$ where $o, d \in M$. We let $(o, v_1, \ldots, v_T, d)$ be a path through the induced subgraph. It is clear that it is not possible to find a better (o, d)-path in the graph than one with value $z(N)$, thus duplicating this path Q times will yield a *lower bound* solution i.e. we have that $Qz(N) \leq P^*$.

An *upper bound* can be obtained by iteratively removing nodes from the graph. The method begins with $M_1 = N$ and iteration counter $i := 1$. The shortest (o, d)-path problem on the subgraph induced by M_i is solved to obtain the path $(o, v_1^i, \ldots, v_T^i, d)$. If $v_t^i = w_{jt} \in \mathcal{W}$, then remove v_t^i as well as all nodes w_{ht} with $(j, h) \in \mathcal{K}$ from M_i to obtain M_{i+1}. We repeat this process if $i \leq Q$, otherwise we terminate with an upper bound value $z(M_1) + \ldots + z(M_Q)$. This is indeed an upper bound as we only construct paths which are compatible with the previously obtained paths. That is, by construction we have that

$$Qz(N) \leq P^* \leq z(M_1) + \ldots + z(M_Q)$$

if the sequence $M_1, \ldots, M_Q$ is obtained using the method described above. Note that we do not exclude any nodes from E, and it is therefore always possible to use the path $(o, e_1, \ldots, e_T, d)$ having the cost of zero. Consequently, the heuristic construction will be trivially upper bounded by zero.

4.4 Comments on Shortest Path algorithms in Acyclic Graphs

In the following sections we use shortest path algorithms extensively, and a few comments are needed prior to this discussion. First of all, recall that the networks in this paper are all acyclic networks which make the algorithms for solving shortest path problems efficient (polynomial worst case time complexity in the number of arcs). Now, suppose that the acyclic graph is given by $G(N, A)$, where N is the set of nodes and A is the set of arcs. The shortest path problem can be solved easily by first sorting the nodes in topological order, and then identifying the shortest path to each of the nodes in this order. See Cormen et al. [2001] or Ahuja et al. [1993] for introductions to solving shortest path problems. We will refer to this approach as a forward pass where we identify the shortest (o, v)-path for $v \in N$. In later sections we use the shortest (v, d)-path for any $v \in N$. This is also easy to identify as we simply have to identify the shortest (d, v)-path for the nodes in a reverse topological order, i.e. we start with d and then extend to the immediate predecessors of d to get their shortest paths. For each of these nodes we would then extend to their predecessors and so forth.

5 Problem Size Reduction

The SMCPSP is highly structured, and we can exploit this structure to assist the exact solution procedure. We provide two ways of simplifying the problem. The first way is to use the single job network to identify jobs that will never be started nor ended in some periods. This is described in Section 5.1. In the second approach, which is described in Section 5.2, we use the single job network for different jobs to compare these and identify which jobs are better than others.

5.1 Single Job Preprocessing

A (u_0, u_{T+1})-path in the single job network described in Section 4.1 gives which periods the job j should be processed. The value $\delta_j(u_0, u_{T+1})$ of a least cost (u_0, u_{T+1})-path in this graph is the best possible contribution of job j. The path from u_0 to u_{T+1} using only nodes from U will have cost zero, and if the least cost (u_0, u_{T+1})-path has value zero then we will never include job j in the optimal solution of the SMCPSP. The reason is that we are indifferent to whether or not to include the job penalty wise. But if we include the job j, then it might use server capacity which is better used for jobs yielding negative penalties. In general, we will always remove paths yielding a zero penalty.

The above argument will be used repeatedly to remove paths – thereby removing the corresponding (partial) jobs – which yield a zero penalty, as they may use server capacity that can be better used for other jobs. Note that we may in some cases be able to include the job without pushing other jobs out of the solution. In that case, we have alternative optimal solutions, where we select the one that excludes jobs yielding a zero total penalty. Thus the preprocessing below aims at identifying a solution excluding as many zero total penalty jobs as possible.

In the case where the least cost (u_0, u_{T+1})-path is negative it may be prudent to include either the full job or parts of the job in the solution. We can, however, identify periods for which we will never start, finish, or continue the job.

Given an arc (n, m), where either $n \in V$ or $m \in V$ with cost c^j_{nm}, it is easy to find a least cost (u_0, n)-path as well as a least cost (m, u_{T+1})-path. Let the values of these least cost paths be $\delta_j(u_0, n)$ and $\delta_j(m, u_{T+1})$, respectively. The value of the least cost path including the arc (n, m) must then be $\delta_j(u_0, n) + c^j_{nm} + \delta_j(m, u_{T+1})$ and if this value is non-negative, then the arc will never be used in the optimal SMCPSP solution we are seeking. However, we can do better than this, as we can eliminate certain arcs which may be on paths yielding negative costs. This elimination is described in the following two propositions.

Proposition 1. *Let $j \in \mathcal{J}$ and suppose that for some $t : a_j < t \leq b_j$ the penalty $p_{jt} > 0$. Then it will never optimal to start job j in period t.*

Proof. If we have to start the job in period t, then the path in the graph has to pass through the node u_{t-1}. From this node two paths exist to node u_{t+1}, one passing through v_t, and one passing through u_t. Comparing the costs of these two paths, we have that $c_{u_{t-1},u_t} + c_{u_t,u_{t+1}} = 0 < c_{u_{t-1},v_t} + c_{v_t,u_{t+1}} = \pi^s + p_{jt} + \pi^e$ for $p_{jt} > 0$. Hence, it is not optimal to pass through v_t on the path from u_{t-1} to u_{t+1}. Next, from the node u_{t-1} two paths to v_{t+1} exist, one passing through u_t, and one passing through v_t. Again comparing the two paths costs we have $c_{u_{t-1},u_t} + c_{u_t,v_{t+1}} = \pi^s + p_{j,t+1} < \pi^s + p_{jt} + p_{j,t+1} = c_{u_{t-1},v_t} + c_{v_t,v_{t+1}}$, and it will again not be optimal to pass through v_t if $p_{jt} > 0$. □

The consequence of Proposition 1 is that if we start a job late, it will always be better to delay starting the job until the penalty is non-positive, and we can therefore eliminate the possibility of using the arc (u_{t-1}, v_t). As we can eliminate the possibility of starting a job late in a period, we can also eliminate the possibility of prematurely ending a job in a period. This is what the next proposition states:

Proposition 2. *Let $j \in \mathcal{J}$ and suppose for some $t : a_j \leq t < b_j$ the penalty $p_{jt} > 0$. Then it will never be optimal to end job j between period t and period $t+1$.*

Proof. Two partial paths exist which end job j from period t to period $t+1$. These paths are $v_{t-1} \to v_t \to u_{t+1}$ and $u_{t-1} \to v_t \to u_{t+1}$. Now, observe that the first path is worse than the path $v_{t-1} \to u_t \to u_{t+1}$ because $c_{v_{t-1},u_t} + c_{u_t,u_{t+1}} = \pi^e < p_{jt} + \pi^e = c_{v_{t-1},v_t} + c_{v_t,u_{t+1}}$, and the second path is worse than $u_{t-1} \to u_t \to u_{t+1}$ as $c_{u_{t-1},u_t} + c_{u_t,u_{t+1}} = 0 < \pi^s + p_{jt} + \pi^e = c_{u_{t-1},v_t} + c_{v_t,u_{t+1}}$. □

As a consequence of Proposition 2, it will be better to end the job earlier than time t or not to end it between time t and time $t+1$ than to end it between the periods t and $t+1$. Thus, we will never use the arc (v_t, u_{t+1}) and it can be eliminated.

When eliminating arcs, we reset their cost values to infinity, i.e. if we eliminate arc (n, m) for job j, we put $c^j_{nm} = \infty$. We will use this to distinguish between the arcs that are usable and the arcs which are not usable in the dynamic programming procedure, which is described in Section 6.

The elimination of arcs described above will tend to become stronger as the penalties of the individual job-time combinations on average get larger. Furthermore, if the penalties π^s and π^e increase, the elimination of arcs will also become stronger, as more shortest paths using arcs having these penalties will become non-negative. This is verified by the computational experiments discussed in Section 7.2.

5.2 Job Ranking and Elimination

The capacity in each period is limited by Q, and it will not be optimal to undertake jobs for which Q better jobs exist. In this section we describe how to rank the jobs such that a job guaranteeing to yield a worse solution if it is undertaken without undertaking another better job will be ranked worse than the other job. Intuitively, given two jobs $i, j \in \mathcal{J}^{cat}_c$, job i is better than job j if for any way of processing job j we can identify a way of processing job i which is at least as flexible and has no larger cost than the way of processing job j. We will exploit the fact that processing sequences are the least flexible ways of processing a job in a specific period. This allows us to make direct comparisons on cost as all other paths are no less flexible than the processing sequences. We formalize the job ranking below.

In this section we only compare jobs within the same category. This is due to the fact that even though a job from one category is better than a job in another category, it may happen that the second category overall has better jobs than the first category. Hence, if we first rank jobs across categories and later eliminate jobs based on this ranking, we may end with a non-optimal solution.

Given a (u_t, u_s)-processing sequence for job $j \in \mathcal{J}^{cat}_c$, then if another job $i \in \mathcal{J}^{cat}_c$ exists with $\alpha_{jts} > \delta_i(u_t, u_s)$, it is better to use the shortest path yielding the value $\delta_i(u_t, u_s)$ for i than to use the (u_t, u_s)-processing sequence for job j. Not only is the cost of using the shortest path for job i less than the cost of using the (u_t, u_s)-processing sequence for job j, but it may also be more flexible than the (u_t, u_s)-processing sequence, because it may

have more elements from U than the processing sequence. Furthermore, if $\alpha_{jts} = \delta_i(u_t, u_s)$ and $i < j$, then for tiebreaking we say that it is better to use the path yielding $\delta_i(u_t, u_s)$ for job i than using the (u_t, u_s)-processing sequence for job j. Finally, if $\alpha_{jts} = \delta_i(u_t, u_s)$ and a $(u_t, w^i_{t+1}, \ldots, w^i_{s-1}, u_s)$ exists having value $\delta_i(u_t, u_s)$ with $w^i_r \in U$ for some $r \in \{t+1, \ldots, s-1\}$, then the path $(u_t, w^i_{t+1}, \ldots, w^i_{s-1}, u_s)$ has the same value for job i as the (u_t, u_s)-processing sequence while being more flexible, as it has more elements from U than the processing sequence. As a consequence, the path $(u_t, w^i_{t+1}, \ldots, w^i_{s-1}, u_s)$ gives the possibility of including another job with negative penalty in period r. Observe that if $\alpha_{jts} \geq 0$, then we can exploit the first and the last conditions, as any other job will have $\delta_i(u_t, u_s) \leq 0$. If $\delta_i(u_t, u_s) < 0$, then we have found a sequence which has a strictly lower cost. On the other hand, if $\delta_i(u_t, u_s) = 0$, then we know that the non-processing sequence $(u_t, u_{t+1}, \ldots, u_s)$ using only nodes from U will have cost 0, which satisfies the last condition. Hence, we are only interested in the situations where $\alpha_{jts} < 0$.

In order to exploit the fact that one job can be considered better than another job, we define three sets of jobs which are regarded as better than job j if we only observe the period from t to s. First we define the set of strictly better jobs as

$$D^1_{jts} = \{i \in \mathcal{J}^{cat}_c | \alpha_{jts} > \delta_i(u_t, u_s)\}$$

Next, the set of jobs that are only being more flexible, but which have the same cost as job j, is given by

$$D^2_{jts} = \{i \in \mathcal{J}^{cat}_c | \alpha_{jts} = \delta_i(u_t, u_s), \exists r : t < r < s \wedge w^i_r \in U\}$$

where w^i_r is a part of a path $(u_t, w^i_{t+1}, \ldots w^i_s, u_s)$ that has a cost equal to $\delta_i(u_t, u_s)$ for job i. Finally, the set of jobs that have the same cost, but which have smaller indices, is defined as

$$D^3_{jts} = \{i \in \mathcal{J}^{cat}_c | \alpha_{jts} = \delta_i(u_t, u_s), i < j\}$$

The latter set is used for tie breaking. Now, the set of jobs which are better than the (u_t, u_s)-processing sequence for job j is the union of the three sets above

$$D_{jts} = D^1_{jts} \cup D^2_{jts} \cup D^3_{jts} \tag{14}$$

When the size of the set D_{jts} increases, there are more jobs which are better than job j for the period pair t, s. Intuitively, if the size of D_{jts} is sufficiently large for all pairs of t, s, then it is less likely that job j will be included in an optimal solution for the SMCPSP. This gives rise to the following proposition for eliminating jobs:

Proposition 3. *Let $j \in \mathcal{J}^{cat}_c$ for category $c \in \mathcal{C}$ and let D_{jts} be defined as in (14). If $|D_{jts}| \geq Q$ for all pairs $t, s \in \mathcal{T}$ with $a_j - 1 \leq t < s \leq b_j + 1$ and $\alpha_{jts} < 0$, then an optimal solution for SMCPSP exists which does not include job j.*

Proof. Suppose that $j \in \mathcal{J}^{cat}_c$ for $c \in \mathcal{C}$ and for one pair t, s with $a_j - 1 \leq t < s \leq b_j + 1$ the processing sequence $(u_t, v_{t+1}, \ldots, v_{s-1}, u_s)$ for job j is a part of the optimal solution $\mathcal{S}^* = (\mathcal{S}^*_1, \ldots, \mathcal{S}^*_T)$. As argued above, if $\alpha_{jts} \geq 0$, then the processing sequence can be replaced by the sequence $(u_t, u_{t+1}, \ldots, u_{s-1}, u_s)$ having the same or better cost than the processing sequence for job j. Hence, assume that $\alpha_{jts} < 0$ and assume that $|D_{jts}| \geq Q$. Then at least one $i \in D_{jts}$ must exist which is not part of the optimal solution. This job i can be in one or more of the sets D^1_{jts}, D^2_{jts}, and D^3_{jts}. If $i \in D^1_{jts}$, then we can replace the processing sequence by a better sequence for job i, and the solution would therefore not be

optimal. On the other hand, if $i \in D^2_{jts} \cup D^3_{jts}$, then a feasible sequence for job i exists which yields the same cost, and we can therefore replace the processing sequence for job j with the sequence for job i. Hence, in any case we can replace the processing sequence for job j with a sequence for job i, which is no worse. As this is true for any solution including the processing sequence for job j, it is also true for an optimal solution including the processing sequence for job j. Hence, we can identify an alternative solution excluding job j. □

Proposition 3 enables us to eliminate all jobs satisfying the condition before solving the actual problem, thus reducing the size of the problem. However, one must exercise caution when using Proposition 3. This is due to elimination based on the set D^2_{jts}, which may result in two jobs both being deemed better than the other, hence risking the possibility of eliminating both jobs. As a consequence, we apply the simpler version where we limit D_{jts} to the union of D^1_{jts} and D^3_{jts} and thereby avoid the problem altogether.

The remaining jobs are then included when solving the problem. These jobs can be ranked depending on the period, i.e. identifying which jobs are best to start in a period and which jobs are best to end in a period. Hence, in a given period, we first rank the jobs which can be started in the period (see Proposition 4) and then we rank the jobs which can be ended in a period (see Proposition 5). This is done by comparing (partial) processing sequences beginning in the period for one job j with the corresponding shortest path value for another job i. We may thereby show the existence of better paths, both cost wise and flexibility wise, for job i for each of the (partial) processing sequences for the first job j.

We begin with the start ranking for a period. If a job j is started in period t, then a (u_{t-1}, u_s)-processing sequence has to be undertaken for job j for some $s > t$. However, if we have another job i which can also be started in period t and for which we can construct a path for arbitrary $s > t$ including the arc (u_{t-1}, v_t) that has a cost no larger than using the (u_{t-1}, u_s)-processing sequence for job j but may be more flexible than the (u_{t-1}, u_s)-processing sequence, then we would start job i instead of job j. Hence, if we start job j in period t, then we have to ensure that job i is also started in period t or has been started in a previous period. This is the main idea in Proposition 4.

Proposition 4. *Let $j \in \mathcal{J}^{cat}_c$ for category $c \in \mathcal{C}$, and for each $i \in \mathcal{J}^{cat}_c$ let $(v_t, w^i_{t+1}, \ldots, w^i_{s-1}, u_s)$ for $v_t \in V$ and $u_s \in U$ be a path having value $\delta_i(v_t, u_s)$. If*

$$\alpha_{j,t-1,s} \geq \delta_i(v_t, u_s) + c^i_{u_{t-1}, v_t} \tag{15}$$

for all $t < s \leq b_j + 1$ with $\alpha_{j,t-1,s} > \delta_i(v_t, u_s) + c^i_{u_{t-1}, v_t}$ for some s, then it will never be better to start job j in period t without either starting job i in period t or processing job i in both the periods $t-1$ and t.

Proof. Suppose that jobs $i, j \in \mathcal{J}^{cat}_c$ and suppose that (15) holds for all $t < s \leq b_j + 1$. In the case where job i is not being processed in period $t-1$ and period t, we have the following: If we start job j, then regardless of which time s we end job j we can identify a sequence for job i which yields no more penalty than job j. Hence it will not be better to start job j without starting job i. If job i is being processed in the periods $t-1$ and t, then it will take up a position and only $Q-1$ positions are left for job j, in which case we can freely start job j because the better job i is already started. □

Proposition 4 can be used to rank the jobs for starting. If (15) holds as an equality for all $t < s \leq b_j$, then the two jobs can be viewed as equally good, and we then rank on the job index, i.e. we say that it is better to start job i than job j if $i < j$. Now let $j \in \mathcal{J}^{cat}_c$ and

let $\mathcal{R}^s_{jt} \subseteq \mathcal{J}^{per}_t \cap \mathcal{J}^{cat}_c$ be the set of jobs i satisfying Proposition 4 as well as the jobs with a lower index satisfying (15) with equality for all $t < s \leq b_j$, i.e. it is better to start each job $i \in \mathcal{R}^s_{jt}$ in period t than starting job j in period t. Clearly, we have that if $|\mathcal{R}^s_{jt}| \geq Q$, then we will never start job j in period t.

Now we turn to the ranking of jobs which can be ended in a period. The idea is that if we have (at least) two jobs, i, j, receiving service from the previous period and we are trying to end one of the jobs, say job j, prematurely, then if job i can obtain a better cost than job j while being more flexible in the upcoming periods, we will rather end job j than job i. Hence, we would either end both jobs, end job j, or continue both jobs. This is basically what Proposition 5 states.

Proposition 5. *Let $j \in \mathcal{J}^{cat}_c$ for category $c \in \mathcal{C}$ and for each $i \in \mathcal{J}^{cat}_c$ let $(v_t, w^i_{t+1}, \ldots, w^i_{s-1}, u_s)$ for $v_t \in V$ and $u_s \in U$ be a path having value $\delta_i(v_t, u_s)$ with $b_i \leq b_j$. If*

$$\beta_{j,t-1,s} \geq \delta_i(v_t, u_s) + c^i_{v_{t-1},v_t} \tag{16}$$

for all $t < s \leq b_j + 1$ with $\beta_{j,t-1,s} > \delta_i(v_t, u_s) + c^i_{v_{t-1},v_t}$ for some s, then it will never be better to continue job j instead of job i in period t without continuing job i in period t.

Proof. Suppose that $\mathcal{S}^* = (\mathcal{S}^*_1, \ldots, \mathcal{S}^*_T)$ is an optimal solution and $i, j \in \mathcal{S}^*_{t-1}$ are included in the optimal solution in period t. Furthermore, suppose that jobs i, j are defined as in Proposition 5. If we continue job j and discontinue job i from period $t-1$ to period t, we incur the penalty $\beta_{j,t-1,s}$ for some future period s where job j is discontinued. However, for this period we could use $(v_{t-1}, v_t, w^i_{t+1}, \ldots, w^i_{s-1}, u_s)$ for job i, which yields a cost $\delta_i(v_t, u_s) + c^i_{v_{t-1},v_t}$ that is no larger than job $\beta_{j,t-1,s}$. Thus, we can always exchange the partial processing sequence $(v_t, \ldots, v_{s-1}, u_s)$ for job j with the corresponding best path for job i and thereby get a solution which is no worse. □

As for the insertions we can establish the set of jobs which are better to end after period $t-1$ compared to a given job j. We denote as $\mathcal{R}^e_{jt}$ the set of all the jobs i satisfying Proposition 5 as well as the jobs $i < j$ satisfying (16) with equality for all $t < s \leq b_j + 1$. Note that we cannot use the relation $|\mathcal{R}^e_{jt}| \geq Q$ to state that we will never end job j in period t. This is due to the fact that not all elements of $\mathcal{R}^e_{jt}$ may be present in the set $\mathcal{S}_{t-1}$, and therefore it may still be of interest to end job j after period $t-1$.

As mentioned in the beginning of this section, the ranking and elimination of jobs is only feasible for jobs within the same category. As a consequence, we have that if the number of categories increases while the number of jobs and the capacity remain the same, then ranking and elimination become weaker as we do not have as many jobs in each category to use for the ranking.

6 A Dynamic Programming Approach

The SMCPSP has a structure which lends itself to dynamic programming. In each period we have to identify a combination of at most Q, compatible jobs and we can construct this combination without any restrictions except the cost, which is calculated based on the combination of jobs in the surrounding periods. In terms of dynamic programming we can use the periods as stages and the possible combinations of the jobs in each of the periods as the states. In this section we will use the terms *stage* and *period* interchangeably.

The dynamic programming is based on the construction of feasible combinations of jobs, $\mathcal{S} \in \mathcal{F}_t$, in increasing periods, t. If necessary, we use subscript t for $\mathcal{S}_t$ to explicitly state that the set $\mathcal{S}_t$ is from $\mathcal{F}_t$. We term a feasible combination of jobs $\mathcal{S}$, indicating a state set. In this section we will use the following:

- $P_t(\mathcal{S})$ is the minimal cost of obtaining $\mathcal{S}$ in stage t using any sequence of sets $(\mathcal{S}_1, \ldots, \mathcal{S}_{t-1}) \in \mathcal{F}_1 \times \ldots \times \mathcal{F}_{t-1}$.
- $\mathcal{L}_t(\mathcal{S}) = (P_t(\mathcal{S}), \mathcal{S})$ is the state corresponding to state set $\mathcal{S}$ at stage t, where $\mathcal{S} \in \mathcal{F}_t$.
- $\overline{\mathcal{S}}_t = \mathcal{S}_t \cap \mathcal{J}^{per}_{t+1}$ is the set of jobs included in $\mathcal{S}_t$ in stage t which are not required to finish between period t and period $t+1$, i.e. $b_j > t$ for $j \in \overline{\mathcal{S}}_t$ and $b_j = t$ for $j \in \mathcal{S}_t \setminus \overline{\mathcal{S}}_t$. We will refer to the set $\overline{\mathcal{S}}_t$ as the reduced state set.
- Λ_t is the set of efficient states in stage t. A state is efficient if it is not dominated by any other state in the stage. Sufficient criteria for dominance are described in Section 6.1.
- $\mathcal{J}^s_t = \{j \in \mathcal{J} | c^j_{u_{t-1}, v_t} < \infty\}$ is set of jobs which can be started in period t.
- $\mathcal{J}^e_t = \{j \in \mathcal{J} | c^j_{v_{t-1}, u_t} < \infty\}$ is the set of jobs which can be discontinued after period $t-1$.

We elaborate on some of these elements in what follows. To identify the smallest possible value of the objective function for a state set $\mathcal{S}$, we reformulate the objective (3) as a recursion

$$P_t(\mathcal{S}) = \begin{cases} \sum_{j \in \mathcal{S}} p_{jt}, & t = 1 \\ \sum_{j \in \mathcal{S}} p_{jt} + \min_{\mathcal{S}' \in \mathcal{F}_{t-1}} \left\{ P_{t-1}(\mathcal{S}') + \pi^s f_{t-1,t}(\mathcal{S}', \mathcal{S}) + \pi^e g_{t-1,t}(\mathcal{S}', \mathcal{S}) \right\} & t > 1 \end{cases} \tag{17}$$

where the value of state $\mathcal{S}$ is calculated as the direct penalty received for the state and the minimum penalty of shifting from a set of jobs in the previous period $t-1$ to the set of jobs $\mathcal{S}$ in period t.

Proposition 6. *Let $\mathcal{S}_t \in \mathcal{F}_t$ be a feasible combination of jobs in period $t \in \mathcal{T}$. Then $P_t(\mathcal{S}_t)$ derived by recursion (17) yields the minimum penalty for obtaining $\mathcal{S}_t$ for any sequence $(\mathcal{S}_1, \ldots, \mathcal{S}_t) \in \mathcal{F}_1 \times \ldots \times \mathcal{F}_t$.*

Proof. For $t = 1$, $\sum_{j \in \mathcal{S}_1} p_{j1}$ is exactly the cost of using set $\mathcal{S}_1$ in period 1. Now suppose that $t = 2$ and we have found $P_1(\mathcal{S}_1)$ for all $\mathcal{S}_1 \in \mathcal{F}_1$. Let $\mathcal{S}_2 \in \mathcal{F}_2$, and suppose that $P_2(\mathcal{S}_2)$ is not the minimum penalty value for $\mathcal{S}_2$. Then an $\mathcal{S}_1 \in \mathcal{F}_1$ must exist such that $\min_{\mathcal{S}' \in \mathcal{F}_1} \{P_1(\mathcal{S}') + \pi^s f_{1,2}(\mathcal{S}', \mathcal{S}_2) + \pi^e g_{1,2}(\mathcal{S}', \mathcal{S}_2)\} > P_1(\mathcal{S}_1) + \pi^s f_{1,2}(\mathcal{S}_1, \mathcal{S}_2) + \pi^e g_{1,2}(\mathcal{S}_1, \mathcal{S}_2)$. But this is in contradiction to the left-hand side being minimal. The same argument is valid for $t > 2$, and we thereby have that recursion (17) will give the minimum penalty for obtaining any state $\mathcal{S}_t \in \mathcal{F}_t$ for any $t \in \mathcal{T}$. □

Proposition 7. $\min\{P_T(\mathcal{S}) | \mathcal{S} \in \mathcal{F}_T\}$ *is the optimal value for SMCPSP.*

Proof. By proposition 1 we know that $P_T(\mathcal{S})$ is the minimum value for obtaining state $\mathcal{S} \in \mathcal{F}_T$. Then selecting the minimum value feasible end state, $\mathcal{S}^*$, must be the optimal solution. If it is not, then another solution must exist such that $\mathcal{S}' \in \mathcal{F}_T$ yielding a better end state solution, but this is in contradiction with $\mathcal{S}^*$ having minimal value. □

From this recursion a dynamic programming algorithm can be derived, where we iterate through each stage and generate the possible combinations based on the states from the previous stage. We will elaborate further on the dynamic programming algorithm in Section 6.2.

6.1 State Dominance

A vital part of a dynamic programming algorithm is the dominance used to reduce the number of possible, efficient states. Given two states in the same stage, we say that the first state dominates the second state if we can prove that for any extension of the second state to the end stage an extension of the first state to the end stage exists such that the extension of the first state will have no larger cost than the extension of the second state. In the SMCPSP it is not possible to dominate directly on the cost component as the future cost is dependent on the jobs in the current state. In this section we will present two sufficient conditions for dominance.

The first type of dominance assumes that for each reduced state set $\overline{\mathcal{S}}$ we know a lower bound on the value on any extension of $\overline{\mathcal{S}}$ to the end stage as well as an upper bound on the best feasible extension of $\overline{\mathcal{S}}$. We denote these two bounds as $LB_t(\overline{\mathcal{S}})$ and $UB_t(\overline{\mathcal{S}})$, respectively. Now we have the following dominance rule:

Dominance 1. *If* $\mathcal{S}^1, \mathcal{S}^2 \in \mathcal{F}_t$ *then* $\mathcal{L}(\mathcal{S}^1)$ *dominates* $\mathcal{L}(\mathcal{S}^2)$ *if*

$$P_t(\mathcal{S}^1) + UB_t(\overline{\mathcal{S}}^1) \leq P_t(\mathcal{S}^2) + LB_t(\overline{\mathcal{S}}^2)$$

which is true because the left-hand side is the value of a feasible extension of $\mathcal{S}^1$, whereas the right-hand side is a lower bound on the value of any possible extension of $\mathcal{S}^2$.

The lower bound of the extension of state $\overline{\mathcal{S}}^2$ is easily obtained by using the full period-job network by solving a shortest (d, n)-path problem in the reverse full period-job network for any node n in layer t. Let this shortest path value be γ_n. Then a lower bound on the value of any extension is obtained as

$$LB_t(\overline{\mathcal{S}}^2) = (Q - |\overline{\mathcal{S}}^2|)\gamma_{e_t} + \sum_{j \in \overline{\mathcal{S}}^2} \gamma_{w_{jt}}$$

which is the sum of the shortest path values from d to each node corresponding to job $j \in \overline{\mathcal{S}}^2$ and the shortest path value from d to the empty server node e_t in layer t for each of the unused positions after period t.

An upper bound on the extension of $\overline{\mathcal{S}}^1$ can be derived by any feasible extension of $\overline{\mathcal{S}}^1$ to the end stage. We could modify the approach described in Section 4.3 and not start at the origin node but instead use elements of $\overline{\mathcal{S}}^1$ as the initial nodes for the shortest paths. This may, however, be time consuming as we have to do this for each individual reduced state set. Instead, we rely on a global upper bound for the full problem. If an upper bound solution of value z_{ub} has been found for the full problem, then this bound along with dominance criterion 1 can be used to eliminate states. If a state set $\mathcal{S}$ has $z_{ub} \leq P_t(\mathcal{S}) + LB_t(\overline{\mathcal{S}})$, then it is never possible to reach a solution with lower value than z_{ub} by extending $\mathcal{S}$, and we can therefore eliminate state $\mathcal{L}_t(\mathcal{S})$. Thus, it is of interest to have as tight an upper bound as possible. This approach has the advantage that we do not need to compare every state with every other state, but only with the known upper bound. Hence, the running time is significantly faster; however, we may not eliminate all states that can be removed.

As noted earlier, all state sets in a stage have the same set of feasible extensions. The direct penalty, p_{jt}, of processing a job j in a stage t is independent of the jobs processed in the earlier stages. This is, however, not the case for the late-start penalty π^s and the early-end penalty π^e, where we need to consider the immediate predecessor states. The idea of the next dominance criterion is that we remove all jobs from one state set and instead insert all the jobs from the other state set in the upcoming stage. The extensions of the two state sets will be the same state set. Now, if the first modified state set has a lower penalty than that of the second, unmodified state set, then the second state set will be dominated by the first state set. This is due to the fact that any extension of the second state set can be obtained more cheaply by a similar extension of the modified first state set.

Observe that it is only necessary to take the reduced state sets into consideration because all modifications are done in the next stage. Suppose that we have the two reduced state sets $\overline{\mathcal{S}}^1, \overline{\mathcal{S}}^2$ derived from two state sets $\mathcal{S}^1, \mathcal{S}^2$. To obtain $\overline{\mathcal{S}}^2$ from $\overline{\mathcal{S}}^1$ we have to remove the elements of $\overline{\mathcal{S}}^1 \setminus \overline{\mathcal{S}}^2$ from $\overline{\mathcal{S}}^1$ and insert the elements of $\overline{\mathcal{S}}^2 \setminus \overline{\mathcal{S}}^1$ into $\overline{\mathcal{S}}^1$. In the first case we incur an additional penalty of $\pi^e|\overline{\mathcal{S}}^1 \setminus \overline{\mathcal{S}}^2|$, and in the second case we incur an additional penalty of $\pi^s|\overline{\mathcal{S}}^2 \setminus \overline{\mathcal{S}}^1|$. As we obtain the same state in the next state, the penalties of the individual elements cancel out. Note that in the case where $\overline{\mathcal{S}}^2$ is extended in such a way that elements of $\overline{\mathcal{S}}^2$ are prematurely ended or elements not already in $\overline{\mathcal{S}}^2$ are started, we have that this can be done with no larger penalty (and in some cases with strictly lower penalty) for $\overline{\mathcal{S}}^1$. This gives the following dominance criterion:

Dominance 2. *If $\mathcal{S}^1, \mathcal{S}^2 \in \mathcal{F}_t$, then $\mathcal{L}_t(\mathcal{S}^1)$ dominates $\mathcal{L}_t(\mathcal{S}^2)$ if*

$$P_t(\mathcal{S}^1) + \pi^e|\overline{\mathcal{S}}^1 \setminus \overline{\mathcal{S}}^2| + \pi^s|\overline{\mathcal{S}}^2 \setminus \overline{\mathcal{S}}^1| \leq P_t(\mathcal{S}^2)$$

This criterion has the following implications. First, if $\pi^s = \pi^e = 0$, then it is only necessary to do the following comparison $P_t(\mathcal{S}^1) \leq P_t(\mathcal{S}^2)$, which makes the problem very easy.[1] However, the larger the values of π^s and π^e are, the more difficult it is to use this dominance criterion for dominance. Secondly, an important special case of dominance criterion 2 is when $\mathcal{S}^1$ and $\mathcal{S}^2$ have $\overline{\mathcal{S}}^1 = \overline{\mathcal{S}}^2$; the dominance reduces to comparing the cost of obtaining the states, i.e. $\mathcal{L}_t(\mathcal{S}^1)$ dominates $\mathcal{L}_t(\mathcal{S}^2)$ if $P_t(\mathcal{S}^1) \leq P_t(\mathcal{S}^2)$ and $\overline{\mathcal{S}}^1 = \overline{\mathcal{S}}^2$. Finally, this dominance criterion gives a maximum difference between two efficient states as $|\overline{\mathcal{S}}^1 \setminus \overline{\mathcal{S}}^2| \leq Q$ and $|\overline{\mathcal{S}}^2 \setminus \overline{\mathcal{S}}^1| \leq Q$, i.e. we have that $P_t(\mathcal{S}^2) - P_t(\mathcal{S}^1) < Q(\pi^s + \pi^e)$. Hence, if we keep track of the minimum value of any state obtained in a stage, then we can simplify the dominance check but at the expense of not dominating all possible states. If $P_t^{min} = \min\{P_t(\mathcal{S})|\mathcal{S} \in \mathcal{F}_t\}$, we can eliminate $\mathcal{S}^2$ if $P_t(\mathcal{S}^2) \geq Q(\pi^s + \pi^e) + P_t^{min}$. We apply this version of the dominance check in our dynamic programming algorithm.

6.2 Algorithm

The dynamic programming algorithm is based on the extension of efficient states from one stage to the next. In the following discussion we will assume that we have identified a set of non-dominated states for period $t-1$ (and therefore know the set Λ_{t-1}) and that we are going to construct the set Λ_t.

[1] In fact, in the case where $\pi^s = \pi^e = 0$ we would not solve the dynamic programming, but rather use the approach described in section 2.

Let $\mathcal{L}(\mathcal{S}_{t-1}) \in \Lambda_{t-1}$ and $\mathcal{S}_t \in \mathcal{F}_t$ be a feasible state set for the period t. If $\mathcal{S}_t$ is an extension of $\mathcal{S}_{t-1}$, then $\mathcal{S}_t$ is composed of the jobs carried on from period $t-1$ to period t and jobs newly inserted in period t. We put $\mathcal{O}_t = \mathcal{S}_{t-1} \cap \mathcal{S}_t \subseteq \mathcal{J}_t^{per}$ to be the set of (old) jobs staying in the state set and $\mathcal{I}_t = \mathcal{S}_t \setminus \mathcal{S}_{t-1} \subseteq \mathcal{J}_t^{per}$ to be the set of newly inserted jobs in the state set. Hence, $\mathcal{S}_t = \mathcal{O}_t \cup \mathcal{I}_t$ is partitioned into the set of old jobs and the set of new jobs. We say that $\mathcal{O}_t$ and $\mathcal{I}_t$ are compatible if $\mathcal{O}_t \cup \mathcal{I}_t \in \mathcal{F}_t$ and $\mathcal{O}_t \cap \mathcal{I}_t = \emptyset$. The requirement of $\mathcal{O}_t$ and $\mathcal{I}_t$ being disjoint is to avoid counting the penalty of the intersection of these sets twice. If this penalty is negative, then it would be prudent to add the element in the intersection twice, which should not be possible. Note that any efficient state set in $\mathcal{F}_t$ can be obtained using this partition. The cost of the partition for the continued jobs is given by

$$d_t^o(\mathcal{S}, \mathcal{O}) = P_{t-1}(\mathcal{S}) + \sum_{j \in \mathcal{O}} p_{jt} + |(\mathcal{S} \cap \mathcal{J}_t^{per}) \setminus \mathcal{O}| \pi^e \tag{18}$$

where $d_t^o(\mathcal{S}, \mathcal{O})$ is the penalty incurred when extending $\mathcal{S} \in \mathcal{F}_{t-1}$ while only keeping the jobs $\mathcal{O} \subseteq \mathcal{S}$ in the new state. Observe that the set $\mathcal{O}$ can be obtained from any state set $\mathcal{S}$ containing $\mathcal{O}$, and we can therefore get many different values of $d_t^o(\mathcal{S}, \mathcal{O})$ depending on the state sets $\mathcal{S}$. We are interested only in using the $\mathcal{S}$ for which $d_t^o(\mathcal{S}, \mathcal{O})$ is minimal, and we therefore put the cost of $\mathcal{O}$ to be

$$d_t^o(\mathcal{O}) = \min \{ d_t^o(\mathcal{S}, \mathcal{O}) | \mathcal{L}_{t-1}(\mathcal{S}) \in \Lambda_{t-1}, \mathcal{O} \subseteq \mathcal{S} \} \tag{19}$$

The cost of the newly inserted jobs is calculated as

$$d_t^n(\mathcal{I}) = \sum_{j \in \mathcal{I}} p_{jt} + |\mathcal{I} \cap \mathcal{J}_{t-1}^{per}| \pi^s \tag{20}$$

where $d_t^n(\mathcal{I})$ is the penalty of inserting the jobs $\mathcal{I}$ into a state in period t. The penalty of the composite state set $d_t(\mathcal{O}, \mathcal{I})$ is then upper bounded by $d_t(\mathcal{O}, \mathcal{I}) \leq d_t^o(\mathcal{O}) + d_t^n(\mathcal{I})$. If no job has been removed from $\mathcal{S}_{t-1}$ and reinserted in $\mathcal{I}_t$, then this bound holds as an equality i.e. if $(\mathcal{S}_{t-1} \setminus \mathcal{O}) \cap \mathcal{I} = \emptyset$ then $d(\mathcal{O}, \mathcal{I}) = d_t^o(\mathcal{O}) + d_t^n(\mathcal{I})$. It is, however, always possible to select the partition of $\mathcal{O} \cup \mathcal{I}$ such that we avoid reinsertion of already removed jobs. We can now rewrite the recursion (17) in terms of the sets $\mathcal{O}$ and $\mathcal{I}$ as follows:

$$P_t(\mathcal{S}_t) = \min \left\{ d_t(\mathcal{O}, \mathcal{I}) \,\middle|\, \mathcal{S}_t = \mathcal{O} \cup \mathcal{I}, \mathcal{O} \subseteq \mathcal{S} : \mathcal{L}_{t-1}(\mathcal{S}) \in \Lambda_{t-1}, \mathcal{I} \subseteq \mathcal{J}_t^{per}, \mathcal{O} \cap \mathcal{I} = \emptyset \right\} \tag{21}$$

Hence, it is sufficient to construct the partial state sets $\mathcal{O}$ and $\mathcal{I}$ and merge these into full state sets $\mathcal{S}_t = \mathcal{O} \cup \mathcal{I}$. Thus, in our proposed algorithm, we suggest to construct the sets $\mathcal{O}$ and $\mathcal{I}$ such that we exploit the information from Proposition 4 and Proposition 5 and then merge the compatible pairs to form the state sets $\mathcal{S}_t$. We elaborate on this below.

The sets $\mathcal{O}$ are constructed by removing jobs from state sets $\mathcal{S}$ with $\mathcal{L}_{t-1}(\mathcal{S}) \in \Lambda_{t-1}$. It is only allowed to have elements from $\overline{\mathcal{S}}$ in the set $\mathcal{O}$. Hence the number of newly constructed $\mathcal{O}$-sets is at most $2^{|\overline{\mathcal{S}}|}$. As we have proven that it is never optimal to discontinue some jobs after period $t-1$, we only consider the jobs from $\overline{\mathcal{S}} \cap \mathcal{J}_t^e$ for removal. The number of possible sets $\mathcal{O}$ can be further reduced if we take Proposition 5 into account, i.e. we only remove a job j from $\overline{\mathcal{S}}$ if all jobs i having $i \in \mathcal{R}_{jt}^e \cap \overline{\mathcal{S}}$ are also removed. We let Γ_t be the set of all generated $(d_t^o(\mathcal{O}), \mathcal{O})$.

The construction of the sets $\mathcal{I}$ is slightly more complicated than the construction of the $\mathcal{O}$-sets. First of all, we can only construct sets of jobs in which the jobs are compatible. Hence, in the following, we consider a single category $c \in \mathcal{C}$ and the jobs in $\mathcal{J}' = \mathcal{J}_{ct} \cap \mathcal{J}_t^s$

for which it is feasible to start the job in period t and for which we have not proven that it will never be optimal to start the job in the period t.

Let $(n_1, \ldots, n_k)$ be a sorted sequence of the jobs from $\mathcal{J}'$, i.e. for each $i = 1, \ldots, k$ the job $n_i \in \mathcal{J}'$. The sequence is sorted such that for $1 \leq i < j \leq k$ the job n_j is not ranked better than job n_i, i.e. $n_i \notin \mathcal{R}^s_{n_j t}$. This sorting can be achieved through a topological sorting of the jobs using the ranking as arcs. Now let $\{o, n_1, \ldots, n_k, d\}$ be the set of nodes in a graph having one arc from o to each n_i for $i = 1, \ldots, k$ and one arc from each n_i to each n_j with $i < j$ as well as an arc from each n_i to d. Then we have an acyclic graph where an (o, d)-path containing at most Q nodes from $\{n_1, \ldots, n_k\}$ corresponds to an insertion set, $\mathcal{I}$. Identifying all such paths from the graph corresponds to identifying the set of insertion sets. The number of paths having a length of at most Q is $\sum_{q=0}^{Q} \binom{|\mathcal{J}'|}{q}$, which is polynomial in size for fixed Q, but exponential in size for increasing Q. Hence, it is critical that this size is reduced. The ranking has to be taken into account, which we do next.

We are interested in exploiting Proposition 4 to limit the number of insertion sets generated. It states that if job n_i is ranked better than job n_j then it either has to be started at the same time as job n_j or it has to have been started in an earlier period. Hence, if we include job n_j in $\mathcal{I}$, then a position for job n_i has to be reserved in $\mathcal{O} \cup \mathcal{I}$. Consequently, as we have sorted the jobs according to rank, a path visiting node n_j either has to have visited node n_i as well, or any extension of the path has to include at most $Q - 1$ of the nodes in $\mathcal{J}'$. The first case is where job n_i is in $\mathcal{I}$, while the latter case corresponds to requiring that node n_i is in the set $\mathcal{O}$. In general we must reserve positions for all $n_i \in \mathcal{R}^s_{n_j t}$ which are not a part of the path prior to visiting node n_j. This reservation of positions effectively reduces the number of $\mathcal{I}$-sets constructed. We denote the set of generated pairs as $(d^n_t(\mathcal{I}), \mathcal{I})$ for Θ_{tc}.

After constructing the sets Γ_t and Θ_{tc} it is necessary to merge the elements into states, i.e. to construct $(P_t(\mathcal{S}), \mathcal{S})$ where $\mathcal{S} = \mathcal{O} \cup \mathcal{I}$ for $\mathcal{O} \in \Gamma_t$ and $\mathcal{I} \in \Theta_{tc}$ for $c \in \mathcal{C}$ such that $\mathcal{S} \in \mathcal{F}$. The first requirement is that the number of jobs in $\mathcal{O}$ and $\mathcal{I}$ does not exceed Q. The second requirement is that either $\mathcal{O} = \emptyset$ or $\mathcal{I} = \emptyset$ or all jobs in $\mathcal{O}$ and $\mathcal{I}$ have to be of the same category. The third requirement is that $\mathcal{I} \cap \mathcal{O} = \emptyset$, i.e. the sets do not have any jobs in common. Finally, the fourth requirement is that for each job $j \in \mathcal{I}$ the set $\mathcal{R}^s_{jt} \subset \mathcal{O} \cup \mathcal{I}$, i.e. each of the jobs which are better to insert in period t has to be included in the state set. If any of the four requirements is not satisfied, then the state set $\mathcal{S} = \mathcal{O} \cup \mathcal{I}$ is discarded. On the other hand, if all the requirements are satisfied, then the state set yields a new possible state with cost $d^o_t(\mathcal{O}) + d^n_t(\mathcal{I})$. Clearly, there can be several different ways to construct the same state set $\mathcal{S}$ by merging different elements from Γ_t and Θ_{tc}, and we choose only the state with the least cost. If more than one merging attains this least cost, we choose arbitrarily. In practice, we have a duplicate check based on a key value of the state. This can be implemented using a hashtable.

The dynamic programming procedure is summarized in Algorithm 1. An initial set of efficient states for period 0 is initialized to be a dummy state $(0, \emptyset)$. The algorithm then iterates through the stages. In each stage t the set of Γ_t is constructed by identifying the $\mathcal{O}$-sets from the efficient states in the previous stage. For each category the set of possible new insertions is then constructed by setting up the graph structure described above. After constructing Θ_{tc} the algorithm merges this with the elements from Γ_t. The algorithm prunes inefficient states from Λ_t once all categories have been considered. This is done by applying the dominance relations which are described in Section 6.1. The algorithm has the following time bound:

```
1 Λ_0 ← {(0, ∅)};
2 for t ← 1 to T do
3 |   Γ_t ← Construct_O_sets (Λ_{t−1}, t );
4 |   foreach c ∈ C do
5 |   |   Θ_tc ← Construct_I_sets (t, c);
6 |   |   Λ_t ← Λ_t ∪ Merge (Γ_t, Θ_tc);
7 |   end
8 |   Λ_t ← Eff (Λ_t);
9 end
```

Algorithm 1: SMCPSP dynamic programming

Proposition 8. *The time complexity of algorithm 1 is $O(CTm^Q)$, where $m = \max\{|\mathcal{J}_{ct}||c \in \mathcal{C}, t \in \mathcal{T}\}$.*

Proof. The algorithm runs T iterations and we will bound each iteration. Each iteration can be subdivided into C iterations – one for each category. Hence, we limit our attention to finding the complexity of an iteration for a single period and a single category, and afterwards this complexity has to be multiplied by CT. Consequently, we have to show that each such iteration has complexity $O(m^Q)$.

The largest number of unique state sets which can be generated for any category in any period is $\sum_{q=0}^{Q} \binom{m}{q}$, where $m = \max\{|\mathcal{J}_{ct}||c \in \mathcal{C}, t \in \mathcal{T}\}$. Now note that $\binom{m}{q} \leq \frac{m^q}{q!}$, and we therefore use the upper bound on the binomial coefficient instead of the binomial coefficient itself.

Generating the $\mathcal{O}$-sets is bounded by $O(m^Q)$, which can be realized as follows: The number of state sets of the same category from the previous iteration is $\sum_{q=0}^{Q} \frac{m^q}{q!}$ and from each of these we may generate 2^q new $\mathcal{O}$-sets, of which many may be duplicates. Keeping these in a list or a hashtable makes it possible to replace duplicates in $O(q)$ time, where the check itself is constant, but the length of the state set is q. It costs q to output the individual sets. Let Δ be the constant in the duplicate check, and then the generation of the $\mathcal{O}$-sets takes $\sum_{q=0}^{Q} \frac{\Delta q^2 2^q m^q}{q!}$. Note that $\frac{q^2 2^q}{q!} \leq 12$ for any q and it is therefore constantly bounded. The leading term of this expression is m^Q. Therefore, generating the $\mathcal{O}$-sets is bounded by $O(m^Q)$.

Generating the $\mathcal{I}$-sets has a complexity equivalent to identifying all paths of length at most Q in a graph, which can be done in $\sum_{q=0}^{Q} \frac{m^q}{q!}$ time and each of these sets have to be returned, in total requiring $\sum_{q=0}^{Q} \frac{qm^q}{q!}$ iterations. As $\frac{q}{q!} \leq 1$ for any q we have that generating all $\mathcal{I}$-sets is bounded by $O(m^Q)$.

We can merge $\mathcal{O}$-sets and $\mathcal{I}$-sets as long as the total number of elements do not exceed the capacity Q and they do not have any elements in common. An upper bound on the number of possible merges is

$$\sum_{q=0}^{Q} \sum_{p=0}^{Q-q} \binom{m}{q} \binom{m}{p} \leq \sum_{q=0}^{Q} \sum_{p=0}^{Q-q} \frac{1}{q!p!} m^{q+p} \tag{22}$$

The leading terms of this expression are those where $p + q = Q$ and this leading term is present $Q + 1$ times in the sum above. Furthermore we have that $p!q! \geq \lfloor Q/2 \rfloor! \lceil Q/2 \rceil!$ for

$q + p = Q$. Each of these generated sets must returned and checked for duplicates, which can be done in Q^2 time per set. We have that

$$\frac{Q^2(Q+1)}{\lfloor Q/2 \rfloor ! \lceil Q/2 \rceil !} \leq 9 \tag{23}$$

for any Q, and therefore the merge is upper bounded by $O(m^Q)$.

Each of the three operations above is repeated once in each of the CT iterations. The algorithm is therefore bounded by $O(CTm^Q)$. □

7 Computational Experiments

It is of interest to find out in which cases the dynamic programming procedure performs well and in which cases the mathematical model performs better. To this end we have constructed a set of test instances. These are described in Section 7.1, and the results are discussed in Section 7.2.

The algorithms have been implemented in C++ and compiled with the MinGW 4.5.2 compiler. The tests have been run on a Windows 7 based laptop equipped with an Intel i7 cpu and 8Gb RAM. The MILP model described in Section 2.1 has been implemented using the COIN-OR interface to the CPLEX 12.2 32bit solver. When solving the MILP model we use the default settings for CPLEX. Note that using the default setting for CPLEX allows CPLEX to use more than one thread when solving the MILP model – in our case it can use up to 8 parallel threads.

7.1 Test instances

In order to test the described methods a set of 15 random instances has been constructed. The intention with these tests is to show how the methods behave when we change the number of categories, the size of the penalties π^s and π^e, and finally the capacity of the server.

Each of the instances has 500 jobs and a planning horizon of 25 periods. The duration of each job is a uniformly distributed integer between 1 and 10. All jobs have job-period penalties p_{jt} uniformly distributed in the interval $[-30.0, 10.0]$. This allows for a job to have a possibly positive penalty, while making most jobs profitable. The instances are divided into five subsets with three instances each. These subsets have a different number of categories. The number of categories is 1, 2, 4, 8 and 16. Within each subset three instances are constructed. One where the value of the penalties is low, $\pi^s = \pi^e = 5.0$, one where the value of the penalties is medium, $\pi^s = \pi^e = 25.0$, and one where the value of the penalties is high, $\pi^s = \pi^e = 100.0$. This makes it increasingly undesirable to start a job late as well as end a job prematurely. In order to test the problem for different server capacities we duplicate the 15 sets into 30 sets, where the 15 first have a server capacity of two and the second 15 have server capacity of four. We omit the trivial case having a server capacity of one as this is solved directly by the lower bounding approach described in Section 4.3.

7.2 Results

We have solved each of the instances described in Section 7.1 with a server capacity of two and of four. In this section we will discuss the effects of the preprocessing, as well as compare the methods described previously.

			$Q=2$				$Q=4$			
I	C	T	$\lvert\mathcal{J}\rvert$	avg. EA	avg. $\lvert\mathcal{R}^s_{jt}\rvert$	avg. $\lvert\mathcal{R}^e_{jt}\rvert$	$\lvert\mathcal{J}\rvert$	avg. EA	avg. $\lvert\mathcal{R}^s_{jt}\rvert$	avg. $\lvert\mathcal{R}^e_{jt}\rvert$
1	1	L	109	2.80	6.11	5.85	170	3.06	9.88	9.50
2	1	M	152	3.32	5.62	4.07	218	3.49	8.40	6.17
3	1	H	176	7.48	6.01	2.50	250	7.49	8.71	3.94
4	2	L	170	3.28	5.01	4.85	251	3.38	7.77	7.58
5	2	M	213	3.54	3.66	2.72	303	4.17	6.08	4.72
6	2	H	244	7.55	4.27	1.96	324	8.06	5.93	2.87
7	4	L	248	2.87	3.53	3.39	343	3.09	5.09	4.90
8	4	M	301	4.25	3.03	2.33	370	4.56	3.93	3.03
9	4	H	300	7.87	2.62	1.27	384	8.53	3.72	1.98
10	8	L	336	3.16	2.42	2.34	412	3.24	3.11	3.03
11	8	M	359	4.51	1.87	1.46	412	4.81	2.33	1.86
12	8	H	360	8.29	1.63	0.83	430	8.78	2.10	1.20
13	16	L	405	3.46	1.49	1.45	453	3.41	1.72	1.69
14	16	M	408	4.50	1.08	0.84	446	4.76	1.24	1.00
15	16	H	427	8.66	1.05	0.57	466	9.05	1.26	0.75

Table 1: Effects of the preprocessing. All numbers are rounded to two decimal points.

In Table 1 the effects of the preprocessing are given. Column I is the instance number, C is the number of categories for the instance and T indicates whether the values of π^s and π^e are low (L), medium (M) or high (H). The remainder of the table is divided into two parts; one part for $Q = 2$ and another part for $Q = 4$. Each of these parts has the size of the job set $|\mathcal{J}|$ after job elimination of full jobs. Avg. EA is the average number of arcs eliminated in the single job graph for the jobs remaining after the elimination of full jobs. Finally, avg. $|\mathcal{R}^s_{jt}|$ and avg. $|\mathcal{R}^e_{jt}|$ give, respectively, the average size of the number of jobs which are better to start than a given job and the average size of the number of jobs in a specific period, which are better to finish than a given job in a specific period.

In all cases, the number of jobs is reduced by the preprocessing. The reduction is most significant in the single category case, however. This is not surprising as the elimination based on Proposition 3, becomes weaker as the number of categories increases while the number of jobs remains the same. It is evident that the number of eliminated arcs in the single job graph increases when we increase the value of π^s and π^e. This is mainly due to the shortest path based elimination, as it becomes more likely that the arcs having the larger penalties will never be on a negative penalty path. The average sizes of $\mathcal{R}^s_{jt}$ and $\mathcal{R}^e_{jt}$ tend to decrease when the number of categories increases and when the penalties get larger. This is no surprise either; the average number of jobs in each category will decrease as the number of categories increases (and the number of jobs does not increase by the same magnitude). Hence, there are simply fewer jobs which can be better than a given job in the same category.

We have solved each of the aforementioned 15 instances with a server capacity of 2 and a server capacity of 4. The results of these tests can be seen in Tables 2 and 3, respectively. Each row corresponds to an instance. The first four columns describe instance characteristics, where I is the instance number, $|\mathcal{C}|$ is the number of categories, Q is the server capacity, and T is the type of π^s and π^e with low (L), medium (M), and high (H). The following three columns show the calculated bounds. LB and UB are the shortest path based lower and upper bounds described in Section 4.3, whereas LP is the lower bound derived using the LP relaxation of the MILP model described in Section 2.1. We have not included computation times for these as they are small. Next, the computation time $T_{IP}(s)$

I	C	T	LB	LP	UB	$T_{IP}(s)$	$T_{Dyn}(s)$	P^*
1	1	L	-1275.19	-1256.69	-1252.69	0.73	0.13	-1256.70
2	1	M	-1161.61	-1116.19	-1116.19	2.03	0.16	-1116.19
3	1	H	-1098.53	-1071.15	-1071.15	0.75	0.12	-1071.15
4	2	L	-1265.25	-1223.47	-1212.47	1.40	0.20	-1221.26
5	2	M	-1188.23	-1075.71	-1025.48	1.65	0.28	-1072.24
6	2	H	-1089.84	-995.22	-827.02	7.75	0.20	-964.79
7	4	L	-1298.52	-1229.58	-1217.05	1.65	0.52	-1225.11
8	4	M	-1142.38	-1022.72	-965.73	1.79	0.76	-1017.28
9	4	H	-1083.31	-961.20	-761.80	14.57	0.50	-922.50
10	8	L	-1286.16	-1183.48	-1117.99	2.43	0.79	-1179.11
11	8	M	-1176.10	-968.54	-857.47	22.46	0.67	-912.49
12	8	H	-1166.03	-918.48	-728.78	38.80	1.09	-846.50
13	16	L	-1288.79	-1133.97	-1038.62	1.72	2.27	-1130.70
14	16	M	-1167.01	-909.05	-710.57	51.88	2.27	-846.24
15	16	H	-1156.75	-945.88	-689.68	25.21	1.49	-881.12

Table 2: Results with server capacity $Q = 2$. All numbers are rounded to two decimal points.

in seconds for the integer model using CPLEX is given along with the computation time $T_{DYN}(s)$ in seconds for the dynamic programming. Finally, the optimal value P^* found by the exact approaches is stated.

The shortest path based lower bound LB is in all tests dominated by the LP lower bound. This is not surprising as the LP bound takes the different categories into account while not allowing the solution to use more than one job in each period. In contrast, the shortest path based LB just duplicates the jobs in each period and selects these in the most favorable way. Compared with the LP lower bound we also note that the shortest path based lower bound LB deteriorates as the number of categories increases and the server capacity increases.

The shortest path upper bound is close to the optimal solution when the number of categories is low; however, when the number of categories increases, the distance to the optimal solution also increases. This effect seems to be magnified for the high-penalty case compared to the low-penalty case. Furthermore, the distance from the optimal solution also increases when the capacity of the server increases.

For the MILP the computation time increases as the number of categories increases. The main reason for this is that the integrality gap increases when we increase the number of categories. On the other hand, the computation time of the dynamic programming approach decreases as the number of categories increases. This is due to the decreasing size of $\mathcal{F}_t$ when increasing the number of categories and keeping the number of jobs fixed. It is interesting to see that the dynamic programming approach has the opposite effect in computation time compared to the MILP when we increase the number of categories, and this effect is profound in both directions. For $Q = 4$ and $C = 1$ the MILP model is solved around hundred times faster than the dynamic programming, whereas for $Q = 4$ and $C = 16$ the dynamic programming is solved up 15 times faster than the MILP model.

The M-cases tend to take slightly longer than the L-cases and the H-cases. We believe the reason for this is twofold. When the penalties π^s and π^e decrease, dominance criterion 2 becomes stronger because the part depending on these two penalties decreases, thereby making it easier to dominate a state. On the other hand, when the penalties increase the preprocessing based on Propositions 1 and 2 becomes stronger, removing more arcs from the single job graphs. This will result in significantly fewer constructed states. These two

I	C	T	LB	LP	UB	$T_{IP}(s)$	$T_{Dyn}(s)$	P^*
1	1	L	-2550.38	-2430.54	-2419.58	0.75	77.99	-2430.54
2	1	M	-2323.22	-2084.88	-2081.68	0.76	109.98	-2084.88
3	1	H	-2197.06	-2045.30	-2042.58	0.79	61.24	-2045.30
4	2	L	-2530.50	-2322.89	-2286.64	2.15	23.46	-2318.32
5	2	M	-2376.47	-1924.44	-1696.54	10.98	46.75	-1873.42
6	2	H	-2179.69	-1798.49	-1116.61	7.26	13.18	-1774.00
7	4	L	-2597.03	-2246.75	-2144.46	2.36	8.03	-2238.65
8	4	M	-2284.76	-1821.21	-1612.34	12.60	9.51	-1776.94
9	4	H	-2166.62	-1689.65	-900.02	20.58	2.17	-1607.12
10	8	L	-2572.33	-2133.97	-1825.97	4.81	3.29	-2117.05
11	8	M	-2352.20	-1566.30	-1070.29	46.26	3.17	-1487.36
12	8	H	-2332.06	-1528.51	-760.44	24.97	1.37	-1444.94
13	16	L	-2334.01	-1459.35	-789.21	33.35	2.42	-1379.96
14	16	M	-2577.58	-1878.19	-1421.46	4.82	2.78	-1860.14
15	16	H	-2313.50	-1420.66	-696.68	57.38	3.32	-1278.19

Table 3: Results with server capacity $Q = 4$. All numbers are rounded to two decimal points.

factors each work to the advantage of the algorithms in the extreme cases, whereas they are not significant in the medium case.

8 Conclusion

In this paper we have introduced a new scheduling problem called the Selective Multi-Category Parallel-Servicing Problem. The problem selects (partial) jobs with predetermined processing times for processing on a server capable of executing jobs of the same category in parallel.

The problem can be solved to optimality either by the mixed integer linear programming model presented or by the dynamic programming algorithm constructed. We show that the time complexity for the dynamic programming is polynomial when the capacity of the server is fixed. Furthermore, we have introduced shortest path based methods for identifying lower and upper bounds for the problem. Finally, we introduce several methods for preprocessing the problem.

Our computational study shows that the dynamic programming approach is faster than solving the mixed integer linear programming model when either the capacity of the server is low or the number of categories is high, whereas the contrary is true in the other cases. Furthermore, we demonstrate that the preprocessing significantly reduces the size of the instances.

The problem might be generalized by allowing subsets of categories to be processed simultaneously. This could be interesting if the jobs had more than one characteristic which make them incompatible. We have, however, left this for future research.

References

R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows - Theory, Algorithms, and Applications*. Prentice Hall, 1993.

R. Anbil, C. Barnhart, L. Hatay, E. L. Johnson, and V. S. Ramakrishnan. Crew-pairing optimization at american airlines decision technilogies. In T. A. Ciriani and R. C. Leachman, editors, *Optimization in Industry*, volume 1, chapter 2, pages 31 – 36. John Wiley & Sons, 1993.

E. Andersson, E. Housos, N. Kohl, and D. Wedelin. Crew pairing optimization. In *OR in Airline Industry*. Kluwer Academic Publishers, 1998.

J. Aráoz, E. Fernández, and C. Zoltan. Privatized rural postman problems. *Computers & OR*, 33(12):3432–3449, 2006.

J. Bang-Jensen and G. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer Monographs in Mathematics. Springer, 2001.

D. Black, R. Eglese, and S. Wøhlk. The time-dependent prize-collecting arc routing problem. *Computers and Operations Research*, 40:526 – 535, 2013.

Z.-L. Chen and W. B. Powell. Solving parallel machine scheduling problems by column generation. *INFORMS Journal on Computing*, 11(1):78, 1999.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

P. Demeester, W. Souffriau, P. D. Causmaecker, and G. V. Berghe. A hybrid tabu search algorithm for automatically assigning patients to beds. *Artificial Intelligence in Medicine*, 48:61–70, 2010.

M. Desrochers, J. Desrosiers, and M. Solomon. A new optimization algorithm for the vehicle routing problem with time windows. *Operations Research*, 40(2):342, 1992.

M. Drexl. Rich vehicle routing in theory and practice. *Logistics Research*, 5:47–63, 2012.

D. Feillet, P. Dejax, and M. Gendreau. Travelling salesman problem with profits. *Transportation Science*, 39(2): 188 – 205, 2005.

M. R. Garey. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117 – 129, 1976.

P.C. Gilmore and R.E. Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, 9(6):849 – 859, 1961.

M. Haouri, S. B. Layeb, and H. Sherali. Tight compact models and comparative analysis for the prize collecting steiner tree problem. *Discrete Applied Applied Mathematics*, 161:618 – 632, 2013.

S. Irnich. Resource extension functions: Properties, inversion, and generalization to segments. *OR Spectrum*, 30 (1):113–148, 2008.

S. Irnich and G. Desaulniers. *Shortest path problems with resource constraints*, chapter 2, pages 33–66. Springer: New York, 2005.

D. Pisinger. *Algorithms for Knapsack Problems*. PhD thesis, Department of Computer Science, Copenhagen University, 1995.

T. M. Range, R. M. Lusby, and J. Larsen. A column generation approach for solving the patient admission scheduling problem. Discussion Papers on Business and Economics 1/2013, Department of Business and Economics, University of Southern Denmark, 2013.

K. Smith-Miles and L. Lopes. Measuring instance difficulty for combinatorial optimization problems. *Computers and Operations Research*, 39:875 – 889, 2012.

J. W. Suurballe and R. E. Tarjan. A quick method for finding shortest pairs of disjoint paths. *Networks*, 14: 325–336, 1984.

T. Tholey. *Finding Disjoint Paths on Directed Acyclic Graphs*, volume 3787 of *Lecture Notes in Computer Science*, pages 319–330. Springer, 2005.